공학석사학위논문

선별적 문맥 구분 분석을 위한 예비분석의 자동생성

Generation of Impact Pre-Analysis for Selective Context-Sensitive Analysis

2015 년 8 월

서울대학교 대학원

전기·컴퓨터공학부

강 동 옥

선별적 문맥 구분 분석을 위한 예비분석의 자동생성

Generation of Impact Pre-Analysis for Selective Context-Sensitive Analysis

지도교수 이 광 근

이 논문을 공학석사 학위논문으로 제출함

2015 년 6 월

서울대학교 대학원

전기·컴퓨터공학부

강 동 옥

강동옥의 공학석사 학위논문을 인준함

2015 년 7 월

위 원 장	김명수	(인)
부위원장	이광근	(인)
위 원	신영길	(인)

요약

이 논문은 요약해석[1, 2, 3]에 기반한 선별적 문맥 구분 분석(selective context-sensitive analysis)[4]을 수행할때 이를 위한 예비분석(impact pre-analysis)을 자동으로 생성해주는 방법과 생성을 위해 필요한 입력의 올바름을 검사하는 방법을 제시한다. 선별적 문맥 구분 분석을 위한 예비분석은 여러 알고리즘을 필요로 하기 때문에 설계하고 만드는데 많은 비용이 든다. 또한 대상으로하는 본분석(main analysis)에 따라 예비분석의 디자인이 달라지기 때문에 미리 고정하여 만들 수 없다. 예비분석을 만들 때 본분석에 따라 달라지는 부분은 요약 실행 의미이다. 예비분석의 요약 실행 의미 디자인중 일부를 사용자에게 입력으로 받아 예비분석을 자동생성하는 방법을 소개한다. 이 때 사용자에게 받은 입력이 올바로 설계 되었는지 검사하여 생성된 예비분석이 옳은 입력에서만 동작 하도록 한다. 검사는 입력으로 받은 요약 실행 의미에 대한 정적 분석을 통해 효과적으로 수행할 수 있다. 이를 위해 예비분석의 요약 실행 의미를 기술하는 핵심 언어(core language)와 이에 대한 분석 디자인을 제시한다.

주요어: 요약해석, 정적분석, 프로그램 분석, 선별적 문맥 구분 분석, 자동생성
학번: 2012-20724

목차

그림 목차

표 목차

제 1 장 서론

1.1 동기

이 논문은 요약해석[1, 2, 3]에 기반한 선별적 문맥 구분 분석(selective context-sensitive analysis)[4]을 수행할때 이를 위한 예비분석(impact pre-analysis)을 자동으로 생성하는 방법과 생성을 위해 필요한 입력의 올바름을 검사하는 방법을 제시한다. 선별적 문맥 구분분석이란 정적 분석 수행시에 분석 하고자하는 대상 쿼리(query)를 증명하기 위한 함수 문맥(calling contexts)만을 구분하는 분석이다. 그림 1.1은 버퍼 오버런(buffer overrun) 분석에서 선별적 문맥 구분 분석이 필요한 예이다. 두개의 쿼리중 query1만 문맥 구분을 통해 증명될 수 있다. 따라서 (1)지점의 함수 호출만을 구분해 주는 선별적 문맥 구분 분석이 필요하다. 예비분석이 하는 일은 본분석에서 구분하였을 때 정확도를 향상시키는 함수 문맥을 선별해 주는 것이다. 따라서 대상 언어에 고정된 일반적인 예비분석을 만들 수 없고, 본분석마다 효과적인 예비분석을 디자인하고 구현해야 한다. 이러한 상황에서 예비분석 알고리즘중 본분석과 관련된 부분만을 사용자에게 입력으로 받아 예비분석을 자동으로 생성함으로써 구현하는 비용을 줄일 수 있다. 이때 사용자의 입력이 만족해야하는 조건을 검사해줌으로써 제대로 된 예비분석이 생성되도록 돕는다.

다양한 본분석에 빠르게 선별적 문맥 구분 분석 기술을 적용하기 위해 예비분석의 자동생성이 필요하다. 예비분석을 성공적으로 수행하는 데에는 다음과 같이 여러 가지 계산 모듈이 필요하다. 큰 틀에서 필요한 모듈을 나열해보면, 먼저 본분석의 요약 실행 의미를 효과적으로 요약하는 요약 실행 의미가 필요하다. 그리고 효율적으로 요약 실행 의미의 고정점을 계산해 내기 위해, 값

```
1 int* xmalloc(n){ return malloc(n); }
2
3 int main(){
4   int* p = xmalloc(8);  //(1)
5   int* q = xmalloc(unknown()); //(2)
6   int x = p[1]; //query1
7   int y = q[1]; //query2
8   return x+y;
9 }
```

그림 1.1: 버퍼 오버런 분석에서 선별적 문맥 구분 분석이 필요한 예

흐름 그래프(value flow graph)를 이용하는 고정점 계산 알고리즘을 사용한다. 또 이러한 값 흐름 그래프를 생성하기 위해서 정의된 값 도달 분석(reaching definition analysis)이 필요하다. 정의된 값 도달 분석과 값 흐름 그래프를 이용한 고정점 계산 모듈은 본분석에 따라 크게 변하지는 않지만 구현하는 비용이 많이 든다. 자동생성을 통해 각각을 매 분석마다 디자인하고 구현하는 비용을 절약할 수 있다.

1.2 목표

선별적 문맥 구분 분석(selective context-sensitive analysis)을 위한 예비분석(impact pre-analysis) 생성기는 사용자에게 본분석(main analysis)과 연관된 요소만을 받아서 자동 생성을 수행한다. 본분석과 관련하여 입력으로 받아야하는 요소는 두 가지이다. 첫째는 본분석의 요약 실행 의미를 안전하게 포섭하는 요약 실행 의미이다. 예비분석의 요약 실행 의미가 본분석의 요약 실행 의미를 안전하게 포섭하도록 디자인해야 선별적 문맥 구분 분석을 수행하였을 때 본분석이 예비분석보다 정확한 분석을 수행하게 된다. 따라서 본분석의 요약 도메인을 알고 있는 사용자가 이 부분을 생성기 입력으로 작성해 주어야 한다.

$$\mathbb{V}^\sharp = \{\bot, \star, \top\}$$
$$(\bot \sqsubseteq \star \sqsubseteq \top)$$
$$\gamma(\top) = \mathbb{I}$$
$$\gamma(\star) = \{[a,b] \in \mathbb{I} \mid 0 \leq a\}$$
$$\gamma(\bot) = \emptyset$$

(a) 버퍼 오버런

$$\mathbb{V}^\sharp = \{\bot, \mathtt{N}, \mathtt{Z}, \mathtt{P}, \top\}$$
$$(\bot \sqsubseteq \mathtt{N}, \mathtt{Z}, \mathtt{P} \sqsubseteq \top)$$
$$\gamma(\top) = \mathbb{I}$$
$$\gamma(\mathtt{N}) = \{[a,b] \in \mathbb{I} \mid a \leq -1\}$$
$$\gamma(\mathtt{Z}) = \{[0,0] \in \mathbb{I}\}$$
$$\gamma(\mathtt{P}) = \{[a,b] \in \mathbb{I} \mid a \geq 1\}$$
$$\gamma(\bot) = \emptyset$$

(b) 0으로 나누기

$$\mathbb{V}^\sharp = \{\bot, \bullet, \star, \top\}$$
$$(\bot \sqsubseteq \bullet, \star \sqsubseteq \top)$$
$$\gamma(\top) = \{(i,n) \mid i \in \mathbb{I}, n \in \mathit{Null}\}$$
$$\gamma(\bullet) = \{(i,n) \mid i \in \mathbb{I}, n \in \{\bot, \mathtt{null}\}\}$$
$$\gamma(\star) = \{(i,n) \mid i \in \mathbb{I}, n \in \{\bot, \mathtt{notnull}\}\}$$
$$\gamma(\bot) = \emptyset$$

(c) null포인터 접근

그림 1.2: 선별적 문맥 구분 분석을 위한 예비분석 도메인의 예

그림 1.2는 세 가지 분석 종류별로 본분석의 요약 도메인을 포섭하는 예비분석의 요약 도메인을 디자인 해 본 것이다. (a), (b)는 본분석이 구간 도메인(interval domain) $\mathbb{I}$을 이용하는 분석이다:

$$\mathbb{V} = \mathbb{I}$$

(c)는 본분석이 null분석을 위해 구간 도메인에 포인터 값의 null 여부를 덧붙인 분석이다:

$$\mathbb{V} = \mathbb{I} \times \mathit{Null}$$
$$\mathit{Null} = \{\bot, \mathtt{null}, \mathtt{notnull}, \top\}$$

생성기에서는 이렇게 본분석을 포섭하는 요약 도메인과 그 위에서 작성된 요약 실행 의미를 입력으로 받아서 예비분석을 생성한다. 둘째는 분석 대상 쿼리가 문맥 구분 시 증명 될지 여부를 판단해주는 쿼리 판별기이다. 분석 결과 요약 값이 어떤 모듬 의미(collecting semantics)를 안전하게 포섭하는지는 사용자가 알고 있으므로 쿼리 판별기를 생성기 입력으로 사용자가 작성해 주어야 한다.

두 번째 목표는 예비분석(impact pre-analysis)의 올바른 고정점 계산을 위하여 필요한 조건을 생성기가 검사해주도록 하는 것이다. 선별적 문맥 구분

분석을 위한 예비분석은 모든 문맥을 구분하는 분석이다.[4] 따라서 일반적인 반복 알고리즘으로는 효율적으로 고정점을 계산해 낼 수 없다. 이 한계는 요약 실행 의미가 조인(join) 연산자로만 구성되도록 작성하는 제약을 두고 값 흐름 그래프(value flow graph)를 이용한 분석 알고리즘을 사용함으로써 극복할 수 있다. 생성기는 이 알고리즘을 사용하는 예비분석을 생성하면서, 입력으로 받은 요약 실행 의미가 이 제약사항을 만족하는지 검사하여 만족하지 않을 경우 예비분석을 수행하지 않도록 한다.

1.3 논문의 구성

이 논문은, 총 5장으로 구성되어있다. 2장에서는 선별적 문맥 구분 분석(selective context-sensitive analysis)을 위한 예비분석(impact pre-analysis) 생성기를 설명하고 생성된 분석기의 성능 결과를 다룬다. 3장에서는 생성기에 들어가는 입력 검사기의 디자인 및 검사 예제를 다룬다. 4장에서는 제시하는 생성기의 추후 확장 연구주제를 다루고, 5장에서 결론을 내린다.

제 2 장 선별적 문맥 구분 분석을 위한 예비분석 생성기

이 장에서는 선별적 문맥 구분 분석(selective context-sensitive analysis)을 위한 예비분석(impact pre-analysis) 생성기의 전체 그림과 구체적인 생성 과정에 대해 소개한다. 먼저 생성기가 동작하는 큰 그림을 보인다. 생성기가 대상으로 하는 분석기의 필요조건에 대해 설명하고 생성된 예비분석의 동작을 설명한다. 그리고 생성기에 어떤 입력을 주어야 하는지 구체적으로 기술한다. 마지막으로 사용자의 입력으로 부터 도출하는 정보들과 그것을 가지고 어떻게 예비분석을 생성하는지 보인다.

2.1 큰 그림

2.1.1 생성기가 하는 일

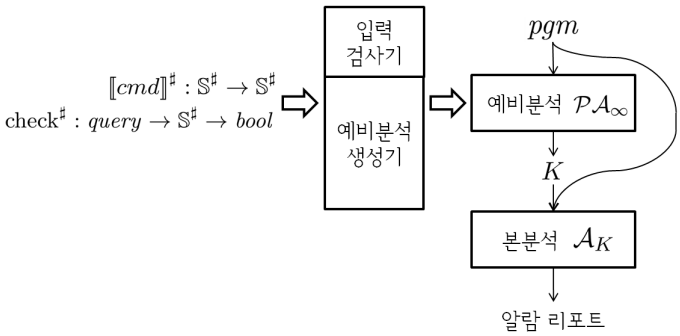

그림 2.1: 선별적 문맥 구분 분석을 위한 예비분석 생성기의 큰그림

사용자가 입력을 주면 생성기는 입력이 적합한지 검사하고 예비분석(impact pre-analysis) $\mathcal{PA}_\infty$을 생성한다. 그림 2.1은 예비분석의 생성 과정과 생성

이후에 예비분석이 본분석(main analysis)에 활용되는 과정을 보여준다. 입력은 크게 두 가지를 받는다. 본분석의 요약 실행 의미를 구성하는 $[\![cmd]\!] : \mathbb{S} \to \mathbb{S}$를 안전하게 포섭하는 $[\![cmd]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$를 받는다. 그리고 분석하려는 쿼리와 요약 메모리 $\mathbb{S}^\sharp$가 주어졌을 때 해당 쿼리를 위해 문맥 구분 분석(context-sensitive analysis)을 하는게 의미 있을지 판별해주는 쿼리 판별기 $\text{check}^\sharp$를 받는다. 입력을 받은 후에 입력 검사기는 $[\![cmd]\!]^\sharp$가 제대로 기술되었는지 검사한다. 이에 대한 구체적인 설명은 3장에서 다룬다. 검사를 통과할 경우 예비분석을 생성해준다.

2.1.2 대상으로 하는 본분석

대상으로 하는 본분석(main analysis)의 핵심 언어(core language)는 다음과 같다.[4] 프로그램은 제어 흐름 그래프(control flow graph)이다:

$$(\mathbb{C}, \to, \mathbb{F}, \iota)$$

여기서, $\mathbb{C}$는 그래프의 노드 집합, $(\to) \subseteq \mathbb{C} \times \mathbb{C}$는 프로그램상의 제어 흐름(control flow) 관계, $\mathbb{F}$는 함수들, $\iota \in \mathbb{C}$는 프로그램 시작지점이다. 각 노드는 명령문을 가지고 있으며 다음과 같다:

$$\begin{array}{lrcl} \text{Command} & cmd & \to & \text{skip} \mid x := e \\ \text{Expression} & e & \to & n \mid x \mid e + e \mid e - e \end{array}$$

본분석은 다음과 같이 구분하는 문맥 K를 인자로 받는 분석이다. 본분석의 요약 도메인과 요약 실행 의미는 다음과 같다:

$$\mathbb{D} = \mathbb{C}_K \to \mathbb{S}$$

$$F(X)(c,\kappa) = [\![\mathsf{cmd}(c)]\!](\bigsqcup_{(c',\kappa') \to_K (c,\kappa)} X(c',\kappa'))$$

$\mathbb{C}_K$는 다음과 같이 정의 된다:

$$\mathbb{C}_K = \{(c,\kappa) \mid c \in \mathbb{C} \wedge \kappa \in K(\mathsf{fid}(c))\}$$

여기서 K는 어떤 함수가 어떤 문맥에서 구분되어야 하는가에 대한 정보이다:

$$K \in \mathbb{F} \to \wp(\mathbb{C}^*_{call})$$

2.1.3 생성한 예비분석의 동작

생성한 예비분석(impact pre-analysis)이 하는 일은 본분석(main analysis)에 구분해야할 문맥을 선택하는 것이다. 먼저 입력으로 받은 $[\![cmd]\!]^\sharp$를 이용하여 다음 요약 실행 의미의 고정점을 구한다. 예비분석의 요약 실행 의미는 모든 문맥(calling contexts)을 구분한다.

$$F^\sharp(X)(c,\kappa) = [\![\mathsf{cmd}(c)]\!]^\sharp(\bigsqcup_{(c',\kappa')\to_\infty(c,\kappa)} X(c',\kappa'))$$

이제 분석하고자 하는 각 쿼리 중 본분석에서 증명될 것으로 보이는 쿼리를 선택해준다. 모든 문맥을 구분하여 구한 요약 실행 의미의 고정점을 가지고 입력으로 받은 $\text{check}^\sharp : \mathit{query} \to \mathbb{S}^\sharp \to \mathit{bool}$ 함수를 이용해 쿼리를 선택한다. 이렇게 선택된 쿼리에 대해서 관련된 문맥들을 모아 본분석에 넘겨준다.

예비분석은 효율적인 분석을 위해 값 흐름 그래프(value flow graph) 위에서 이루어진다.[4] 값 흐름 그래프는 다음과 같이 정의된다.

$$(\Theta, \hookrightarrow_K)$$

여기서,

$$\Theta = \mathbb{C}_K \times \mathit{Var} \quad (\hookrightarrow_K) \subseteq \Theta \times \Theta$$

Θ는 정의(Def)된 지점과 변수를 의미한다. $\hookrightarrow$는 요약 실행 의미에서 어떤 정의(Def)된 변수와 그 변수의 값을 사용(Use)하여 정의(Def)된 변수 사이의 관계이다. 즉, 값이 변수사이에서 어떻게 흘러가는지를 보여준다. 다음과 같이 정의

된다:

$$((c,\kappa),x) \quad \hookrightarrow_K \quad ((c',\kappa'),x')$$

$$iff \quad \begin{cases} (c,\kappa) \to_K (c',k') \wedge x = x' & (\mathsf{cmd}(c') = \text{skip}) \\ (c,\kappa) \to_K (c',k') \wedge x = x' & (\mathsf{cmd}(c') = y := e \wedge y \neq x') \\ (c,\kappa) \to_K (c',k') \wedge x = \mathsf{var}(e) & (\mathsf{cmd}(c') = y := e \wedge y = x') \end{cases}$$

사용자 입력으로 받은 $[\![cmd]\!]^\sharp$는 효율성 조건(efficiency condition)을 만족해야 하는데, 이에 따라 표현식(expression)의 요약 실행 의미 $[\![e]\!]^\sharp$가 변수와 상수의 조인(join)으로만 정의 된다:

$$[\![e]\!]^\sharp(s) = s(x_1) \sqcup \cdots \sqcup s(x_n) \sqcup v$$

따라서 요약 실행 의미의 고정점 계산은 그래프 도달여부(graph reachability) 문제로 환원된다. 값 흐름 그래프 위에서 각 상수 v들을 도달 가능한 모든 지점에 전파하여 조인(join)함으로써 요약 실행 의미의 고정점 값을 구해낼 수 있다.

생성기에서는 이러한 동작을 수행하는 예비분석을 생성한다.

2.2 생성기 입력

2.2.1 요약 실행 의미를 위한 입력

사용자는 요약 메모리 도메인과 cmd에 대한 요약 실행 의미를 정의해 준다. 다음과 같은 요약 메모리 도메인을 입력으로 받는다:

$$\mathbb{S}^\sharp = Var \overset{fin}{\to} \mathbb{V}^\sharp$$

사용자는 이렇게 정의된 도메인 위에서 각 cmd에 대한 요약 실행 의미 $[\![\mathsf{cmd}(c)]\!]^\sharp$를 정의해준다.

입력으로 주는 요약 메모리 도메인과 요약 실행 의미는 효율성 조건(efficiency condition)과 안전성 조건(soundness condition)을 만족해야 한다. 효율성

조건은 반드시 만족해야하는 성질로 3장에서 자세히 설명한다. 안전성 조건의 경우 이를 만족해야 본분석 $\mathcal{A}_K$의 정확도가 예비분석 $\mathcal{PA}_\infty$ 보다 정확하도록 ($\mathcal{A}_K \sqsubseteq_{query} \mathcal{PA}_\infty$) 보장할 수 있다.[4] 입력부분에서 만족해야 하는 안전성 조건은 다음과 같이 $[\![\mathsf{cmd}(c)]\!]^\sharp$가 본분석의 $[\![\mathsf{cmd}(c)]\!]$를 포섭하도록 디자인 하는 것이다:

$$\forall s \in \mathbb{S}, s^\sharp \in \mathbb{S}^\sharp . s \in \gamma(s^\sharp) \Rightarrow [\![\mathsf{cmd}(c)]\!](s) \in \gamma([\![\mathsf{cmd}(c)]\!]^\sharp(s^\sharp))$$

2.2.2 쿼리를 위한 입력

생성기는 값 흐름 그래프(value flow graph)를 이용한 분석을 생성하기 위해 그림 2.1의 check$^\sharp$함수를 두 가지로 나누어 입력받는다.

1. $\mathsf{get\_x} : query \rightarrow Var$
2. $\mathsf{check\_v} : \mathbb{V}^\sharp \rightarrow bool$

get_x는 쿼리에서 검토하고자 하는 변수를 내놓는 함수이다. check_v는 요약 값을 받아서 쿼리를 선택하는 기준에 맞는지 판별해주는 함수이다.

2.3 예비분석 생성 과정

2.3.1 값 흐름 그래프 생성기

예비분석에서 값 흐름 그래프(value flow graph)를 그릴 수 있도록 값 흐름 그래프 생성기를 만들어줘야 한다.

값 흐름 그래프 생성기를 만들기 위해 입력으로부터 요약 메모리 접근정보 $access$를 얻는 함수 get_access를 추출한다. $access$는 다음과 같이 정의된 변수 집합과 사용된 변수 집합의 쌍이다:

$$access = \wp(DefVar) \times \wp(UseVar)$$

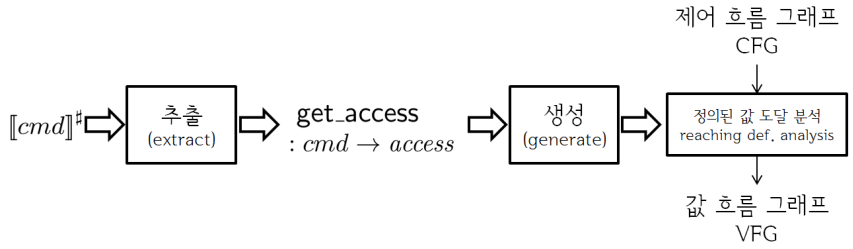

그림 2.2: 값 흐름 그래프 생성기를 생성하는 과정과 예비분석에서의 활용

유저가 입력으로 준 $[\![cmd]\!]^\sharp$을 살펴보면 요약 메모리에서 값을 찾는 부분과 요약메모리에 값을 업데이트 하는 부분을 찾을 수 있다. 이 부분들을 이용하여 cmd별로 $access$정보를 내어주는 get_access 함수를 추출한다.

추출된 함수 get_access를 이용해 정의된 값 도달 분석(reaching definition analysis)을 생성하고 이를 이용하는 값 흐름 그래프 생성기를 만들어낸다. 값 흐름 그래프 생성기는 예비 분석 시에 제어 흐름 그래프(control flow graph)를 받아서 값 흐름 그래프를 만들어주는 역할을 한다. 이러한 값 흐름 그래프를 만들 때 정의된 값 도달 분석이 필요하게 된다. 정의된 값 도달 분석은 정의된 값이 어느 지점에서 사용될 수 있는 지를 알아내는 분석이다. 따라서 정의된 값 도달 분석을 하기 위해서는 각 지점마다 $access$정보를 알고 있어야 한다. get_access는 이러한 $access$정보를 알려주는 역할을 한다. 이렇게 만들어진 정의된 값 도달 분석을 장착하여 값 흐름 그래프 생성기를 만들 수 있다.

2.3.2 상수 발생지점 집합 생성기

상수 발생지점 집합 생성기는 예비분석에서 필요한 상수 발생지점 집합 $\Theta_{v \in \mathbb{V}^\sharp}$을 생성해주는 모듈이다. 값 흐름 그래프(value flow graph) 위에서 분석을 수행할 때 요약 값 별로 다음과 같은 상수 발생지점 집합 $\Theta_{v \in \mathbb{V}^\sharp}$가 필요하다.[4]

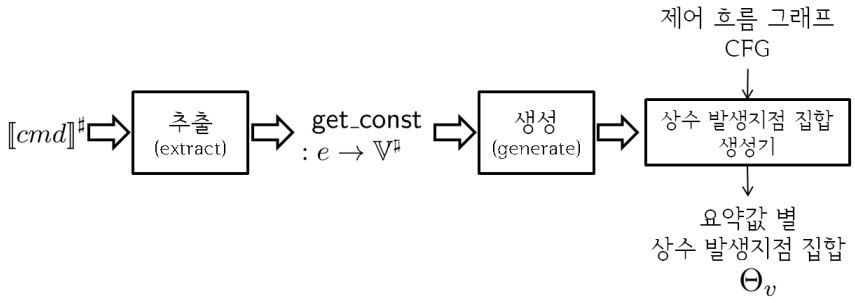

그림 2.3: 상수 발생지점 집합 생성기를 생성하는 과정과 예비분석에서의 활용

$\Theta_{v\in\mathbb{V}^\sharp}$의 정의는 다음과 같다:

$$\begin{aligned}\Theta_{v\in\mathbb{V}^\sharp} \quad = \quad & \{(c,x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{get\_const}(e) = v\} \\ & \cup(\text{if } (v = \top) \text{ then } \{(\iota, x) \mid x \in \mathit{Var}\} \text{ else } \{\})\end{aligned}$$

$\Theta_{v\in\mathbb{V}^\sharp}$ 정의를 보면 상수 발생지점 집합 생성기를 만들 때 get_const함수가 필요하다. 표현식의 요약의미중 상수를 구성하는 부분을 알려주는 함수이다. get_const는 사용자의 입력으로부터 추출한다.

상수 발생지점 집합 생성기를 만드는 데 필요한 get_const함수는 입력 $[\![cmd]\!]^\sharp$으로부터 추출할 수 있다. 표현식에 대한 요약 실행 의미 $[\![e]\!]^\sharp(s) = s(x_1) \sqcup \cdots \sqcup s(x_n) \sqcup v$에서 상수부분 v부분을 찾아서 이를 내놓는 함수를 만들면 된다.

2.3.3 예비분석의 생성

이렇게 입력으로 부터 생성한 것들을 가지고 값 흐름 그래프(value flow graph)를 활용하는 예비분석(impact pre-analysis)을 생성할 수 있다. [4]에 기술된 알고리즘대로 (1)고정점을 구하고 (2)쿼리를 선택하고 (3)선택된 쿼리와 관련된 패스(path)들을 모아 구분할 문맥을 모아주도록 하면 예비분석이 완성된다.

생성기의 입력과 입력으로부터 만들어 낸 값 흐름 그래프 생성기 및 상수

발생지점 집합 생성기는 예비분석의 요약 실행 의미의 고정점 PA_K를 구하는 모듈을 만드는 데에 사용된다. 요약 실행 의미의 고정점 PA_K는 값 흐름 그래프 (value flow graph) $(\Theta, \hookrightarrow_K)$와 상수 발생지점 집합 $\Theta_{v \in \mathbb{V}^\sharp}$를 이용해서 다음과 같이 구할 수 있다:

$$\mathsf{PA}_K(c) = \lambda x. \bigsqcup \{v \in \mathbb{V}^\sharp \mid \exists (c_0, x_0) \in \Theta_v . (c_0, x_0) \hookrightarrow_K (c, x)\}$$

check_v함수는 이러한 고정점 위에서 쿼리의 요약값을 살펴서 문맥을 구분해줄 쿼리를 선택하는 모듈을 만드는 데에 사용된다. get_x로 쿼리의 관심있는 변수를 얻어오고, check_v로 값을 판별해 준다. 실재로는 고정점을 미리 구하지 않고 실시간으로 쿼리의 관심있는 변수에 도달 가능한 상수 발생 지점마다 check_v로 값을 판별한다.

2.4 생성된 예비분석의 성능

표 2.1는 그림 1.2의 (a)에 제시한 도메인을 입력으로 예비분석(impact pre-analysis)을 생성하고 이를 이용하여 선별적 문맥 구분 분석을 수행하였을 때의 결과를 보여준다. 평균 25만 라인 벤치마크 프로그램 9개에 대해 실험하였을 때, 평균 25.2% 알람이 줄었고 62.8%의 비용이 더 들었다.

몇가지 입력을 받아서 자동으로 예비분석을 생성하는 전략이 효과적임을 볼 수 있다. 줄은 알람수는 25.2%로 자동생성 없이 직접 구현한 예비분석[4]에서의 결과와 유사한 본분석의 정확도 향상을 보인다. 분석하는데 증가한 시간은 [4]에서 보인 결과(14%)에 비해 아쉬운 62.8%이 증가하였다. 최적화면에서는 다소 아쉽지만 직접 제작하는 것 만큼의 충분한 정확도 향상을 얻을수 있음을 알 수 있다.

프로그램	라인수	문맥 구분하지 않음		선별적 문맥 구분		알람감소	비용증가
		알람수	시간(초)	알람수	시간(초)	(%)	(%)
spell-1.0	1,596	58	0.7	23	4.2	60.3	83.3
bbe-0.2.2	5,736	436	2.5	202	4.2	53.6	40.4
bc-1.06	11,488	649	27.2	450	116.0	30.6	76.5
unrtf-0.19.3	14,266	1145	24.3	111	68.0	90.3	64.2
tar-1.13	20,606	1249	133.7	1021	168.7	18.2	20.7
trueprint-5.3	20,928	483	12.6	450	16.9	6.8	25.4
agedu-8642	24,111	794	12.4	611	115.3	23.0	89.2
grep-2.5	31,495	755	16.0	702	33.9	7.0	52.8
bison-2.5	101,807	3328	1116.2	3078	3099.1	7.5	63.9

표 2.1: 문맥을 구분하지 않는 분석과 생성된 예비분석을 이용한 선별적 문맥 구분분석의 성능 비교

제 3 장 생성기 입력 검사기

이 장에서는 생성기의 입력 검사기를 어떻게 디자인하는지 설명하고 예제를 통하여 검사기의 동작을 보인다. 생성기의 입력을 기술하는 핵심 언어(core language)를 제시한다. 이 핵심 언어의 의미를 정의하고 요약해석을 기반으로 분석하기 위해 요약 실행 의미를 정의한다. 이 검사기 위에서 검사하고자 하는 성질과 그 성질을 확인하는 전략을 소개한다.

3.1 검사기가 확인하는 성질

3.1.1 검사해야하는 효율 조건

선별적 문맥 구분 분석(selective context-sensitive analysis)을 위한 예비분석(impact pre-analysis)을 효율적으로 분석하기 위한 조건은 아래 세 가지가 있다.[4]

1. 요약 메모리 도메인의 치역인 요약 값 도메인이 유한 완전 래티스 (finite complete lattice):

$$\mathbb{S}^\sharp = Var \xrightarrow{fin} \mathbb{V}^\sharp, \quad (\mathbb{V}^\sharp, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$$

2. 초기 요약 메모리는 $\top$으로 초기화:

$$s_I^\sharp = \lambda x.\top$$

3. 프로그램 표현식 e에 대한 요약 실행 의미가 조인(join) 연산자로만 이루

어져야 한다.

$$\begin{aligned} [\![x := e]\!]^\sharp(s) &= s[x \mapsto [\![e]\!]^\sharp(s)] \\ [\![e]\!]^\sharp(s) &= s(x_1) \sqcup \cdots \sqcup s(x_n) \sqcup v \end{aligned}$$

두 번째 조건은 생성기에서 만족하도록 하면 되는 조건으로 입력과는 상관없는 조건이다. 첫 번째 조건은 입력에 대한 문법적 검사로 쉽게 알아 낼 수 있다. $\mathbb{V}^\sharp$를 정의하는데 사용된 도메인 구조물들이 유한 완전 래티스(finite complete lattice)인지 여부만 살펴보면 된다.

세 번째 조건은 표현식을 계산하는 함수가 조인(join)만 이용하여 값을 내놓는지 판단해야 하는데 단순히 문법적으로 살펴보는 것만으로는 부족하다. $[\![e]\!]^\sharp(s)$를 입력으로 작성할 때 매우 자연스럽게 조인(join) 이외의 연산자 혹은 분기문을 사용하게 된다. 그림 3.1는 프로그램 문법요소 n이 무엇이냐에 따라서로 다른 도메인 원소를 내놓기 위해 분기문을 사용하였다. 이 예제는 분기문이 사용 되었음에도 효율성 조건(efficiency condition)을 만족하는 예제이다. 따라서 단순이 문법형태가 아니라 예비분석의 요약 실행 의미를 잘 따져서 조건을 검사해야 한다.

$$\begin{aligned} [\![n]\!]^\sharp(s) &= \text{if } (n \geq 0) \text{ then } \bigstar \text{ else } \top \\ [\![x]\!]^\sharp(s) &= s(x) \\ [\![e_1 + e_2]\!]^\sharp(s) &= [\![e_1]\!]^\sharp(s) \sqcup [\![e_2]\!]^\sharp(s) \\ [\![e_1 - e_2]\!]^\sharp(s) &= \top \end{aligned}$$

그림 3.1: 효율성 조건을 만족하는 입력 예

3.1.2 검사 아이디어

다음과 같은 조건을 만족하는지 분석하여 표현식(expression)의 요약 실행 의미가 올바로 정의되었는지 검사할 수 있다.

1. 요약 메모리에서 찾은 값은 조인(join)이 아닌 다른 연산자를 만나면 안 된다.
2. 분기문의 조건식에 사용되면 안 된다.

이 두 가지 조건을 확인할 수 있는 정보를 실행 의미에 덧붙여 정의하고 이를 잘 포섭하는 요약 실행 의미를 정의하여 정적분석을 수행하면 이 두 가지 조건을 안전하게 확인할 수 있다.

3.2 핵심 언어 및 정적분석 디자인

3.2.1 핵심 언어

입력을 기술하는 핵심 언어(core language)의 문법은 다음과 같다. $\oplus$는 $\sqcup$ 이외의 모든 연산자를 의미한다. c는 프로그램 문법요소나 기본 값(primitive value), 도메인 원소를 의미하는 상수이다. lookup_l은 요약 메모리에 대한 읽기 접근을 의미한다.

e	$\rightarrow$	$c \mid x \mid \text{lookup}_l$	basic element
	$\mid$	$e \sqcup e$	join
	$\mid$	$e \oplus e$	other op
	$\mid$	if e e e	if branch
	$\mid$	rec f x e	recursive function
	$\mid$	e e	function call

핵심 언어(core language)의 덧붙인 의미(instrumented semantics)는 다음과 같다. *Effect*는 *Val*이 어떤 영향을 받으며 만들어졌는지 알려준다. `con`은

요약메모리 접근이 영향을 미치지 않은 값이다. `lkp`은 요약메모리 접근이 영향을 미친 값이다. `bad`는 요약메모리 접근 이후에 join연산자 이외의 연산자가 영향을 미친 값이다.

$$
\begin{aligned}
v \in \mathit{Val} &= \mathit{Const} \cup \mathit{Bool} \cup \mathit{Proc} \\
\langle \sigma, f, x, e \rangle \in \mathit{Proc} &= \mathit{Env} \times \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \\
\mathit{ef} \in \mathit{Effect} &= \{\mathtt{con}, \mathtt{lkp}, \mathtt{bad}\} \\
\sigma \in \mathit{Env} &= \mathit{Var} \xrightarrow{\mathit{fin}} \mathit{Val} \times \mathit{Effect}
\end{aligned}
$$

$$
\frac{}{\sigma \vdash c \Downarrow (c, \mathtt{con})} \qquad \frac{}{\sigma \vdash x \Downarrow \sigma(x)}
$$

$$
\frac{}{\sigma \vdash \mathrm{lookup}_l \Downarrow (\mathtt{c}_l, \mathtt{lkp})}
$$

$\mathtt{c}_l$는 요약 메모리에서 찾은 값

$$
\frac{\sigma \vdash e_1 \Downarrow (v_1, ef_1) \quad \sigma \vdash e_2 \Downarrow (v_2, ef_2)}{\sigma \vdash e_1 \sqcup e_2 \Downarrow (v_1 \sqcup v_2,\ f_{\sqcup}(ef_1, ef_2))} \qquad \frac{\sigma \vdash e_1 \Downarrow (v_1, ef_1) \quad \sigma \vdash e_2 \Downarrow (v_2, ef_2)}{\sigma \vdash e_1 \oplus e_2 \Downarrow (v_1 \oplus v_2,\ f_{\oplus}(ef_1, ef_2))}
$$

$$
\frac{\sigma \vdash e_1 \Downarrow (\mathtt{true}, ef_1) \quad \sigma \vdash e_2 \Downarrow (v_2, ef_2)}{\sigma \vdash \mathrm{if}\ e_1\ e_2\ e_3 \Downarrow (v_2,\ f_{\mathrm{if}}(ef_1, ef_2))} \qquad \frac{\sigma \vdash e_1 \Downarrow (\mathtt{false}, ef_1) \quad \sigma \vdash e_3 \Downarrow (v_3, ef_3)}{\sigma \vdash \mathrm{if}\ e_1\ e_2\ e_3 \Downarrow (v_3,\ f_{\mathrm{if}}(ef_1, ef_3))}
$$

$$
\frac{}{\sigma \vdash \mathrm{rec}\ f\ x\ e \Downarrow (\langle \sigma, f, x, e \rangle,\ \mathtt{con})}
$$

$$
\frac{\begin{array}{c}\sigma \vdash e_1 \Downarrow (\langle \sigma', f, x, e \rangle, ef) \quad \sigma \vdash e2 \Downarrow (v', ef') \\ \sigma' \cup \{f \mapsto (\langle \sigma', f, x, e \rangle, ef), x \mapsto (v', ef')\} \vdash e \Downarrow (v, ef'')\end{array}}{\sigma \vdash e_1\ e_2 \Downarrow (v, ef'')}
$$

여기서,

$$f_{\sqcup}(ef_1, ef_2) = \begin{cases} \texttt{bad} & \text{if } ef_1 = \texttt{bad} \vee ef_2 = \texttt{bad} \\ \texttt{lkp} & \text{else if } ef_1 = \texttt{lkp} \vee ef_2 = \texttt{lkp} \\ \texttt{con} & \text{otherwise} \end{cases}$$

$$f_{\oplus}(ef_1, ef_2) = \begin{cases} \texttt{bad} & \text{if } ef_1 = \texttt{bad} \vee ef_2 = \texttt{bad} \vee ef_1 = \texttt{lkp} \vee ef_2 = \texttt{lkp} \\ \texttt{con} & \text{otherwise} \end{cases}$$

$$f_{\text{if}}(ef_1, ef_2) = \begin{cases} \texttt{bad} & \text{if } ef_1 = \texttt{bad} \vee ef_1 = \texttt{lkp} \\ ef_2 & \text{otherwise} \end{cases}$$

3.2.2 요약 실행 의미

다음과 같이 요약 실행 의미를 정의하여 안전하게 어떤 값이 요약 메모리에서 찾은 값과 조인(join) 이외의 연산자가 영향을 미쳤는지 판단할 수 있다.[5]

$$\begin{aligned} \hat{v} \in \hat{Val} &= \wp(\hat{Proc}) \\ \hat{\sigma} \in \hat{Env} &= Var \xrightarrow{fin} \hat{Val} \times \hat{Effect} \\ \hat{ef} \in \hat{Effect} &= \{\bot, \hat{\texttt{lkp}}, \top\} \quad (\bot \sqsubseteq \hat{\texttt{lkp}} \sqsubseteq \top) \end{aligned}$$

if, $\oplus$, $\sqcup$에 대한 요약 실행 의미는 다음과 같다. 나머지의 경우 핵심 언어(core language) 의미와 같은 형태로 대체하면 된다.

$$\frac{\hat{\sigma} \vdash e_1 \Downarrow (\hat{v_1}, \hat{ef_1}) \quad \hat{\sigma} \vdash e_2 \Downarrow (\hat{v_2}, \hat{ef_2})}{\hat{\sigma} \vdash e_1 \sqcup e_2 \Downarrow (\hat{v_1} \sqcup \hat{v_2},\ \hat{ef_1} \sqcup \hat{ef_2})}$$

$$\frac{\hat{\sigma} \vdash e_1 \Downarrow (\hat{v_1}, \hat{ef_1}) \quad \hat{\sigma} \vdash e_2 \Downarrow (\hat{v_2}, \hat{ef_2})}{\hat{\sigma} \vdash e_1 \oplus e_2 \Downarrow (\hat{v_1} \hat{\oplus} \hat{v_2},\ \hat{f}_{\oplus}(\hat{ef_1}, \hat{ef_2}))}$$

$$\frac{\hat{\sigma} \vdash e_1 \Downarrow (\hat{v}_1, \hat{ef_1}) \quad \hat{\sigma} \vdash e_2 \Downarrow (\hat{v}_2, \hat{ef_2}) \quad \hat{\sigma} \vdash e_3 \Downarrow (\hat{v}_3, \hat{ef_3})}{\hat{\sigma} \vdash \text{if } e_1\ e_2\ e_3 \Downarrow (\hat{v_2} \sqcup \hat{v_3},\ f_{\text{if}}(\hat{ef_1}, \hat{ef_2} \sqcup \hat{ef_3}))}$$

여기서,

$$\hat{f}_{\oplus}(\hat{ef_1}, \hat{ef_2}) \quad = \quad \begin{cases} \top & \text{if } \hat{\texttt{lkp}} \sqsubseteq \hat{ef}_1 \vee \hat{\texttt{lkp}} \sqsubseteq \hat{ef}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\hat{f}_{\text{if}}(\hat{ef_1}, \hat{ef_2}) \quad = \quad \begin{cases} \top & \text{if } \hat{\texttt{lkp}} \sqsubseteq \hat{ef}_1 \\ \hat{ef}_2 & \text{otherwise} \end{cases}$$

위의 규칙들을 적용할 때 동일한 표현식이 등장할 경우 증명나무에서 동일한 표현식을 가진 바로 직전 조상 노드의 $\hat{\sigma}$를 조인(join)한다. 즉, 문맥을 구분하지 않는 분석(context insensitive analysis)을 수행한다.[5]

$$\begin{array}{c} \hat{\sigma} \sqcup \hat{\sigma}' \vdash e \Downarrow (\hat{v}', \hat{ef}') \\ \vdots \\ \hat{\sigma} \vdash e \Downarrow (\hat{v}, \hat{ef}) \end{array}$$

안전성은 요약해석이론에 기반 하여 쉽게 보일 수 있다. $\hat{Effect}$와 $\hat{Val}$의 γ는 다음과 같다.

$$\gamma_{ef}(\bot) = \{\texttt{con}\} \quad \gamma_{ef}(\hat{\texttt{lkp}}) = \{\texttt{con}, \texttt{lkp}\} \quad \gamma_{ef}(\top) = \{\texttt{con}, \texttt{lkp}, \texttt{bad}\}$$

$$\gamma_v(procs \in \hat{Val}) = \{\gamma_{Proc}(proc) \mid proc \in procs\} \cup Const \cup Bool$$

3.3 검사 결과 예제

그림 3.2은 상수와 요약 메모리 접근이 뒤섞인 입력 예제들이다. (a)는 프로그램 텍스트 n이 0보다 큰지 여부를 판단하여 도메인 원소★, ⊤중 하나를 내놓는다. (b)는 요약메모리에 들어있는 값이 래티스(lattice) 상에서 •보다 낮은 부분 순서(partial order)를 가지는지 판단하여 도메인 원소 ★, ⊤중 하나를 내놓는다. (c)는 (a)와같이 상수만 관련된 분기를 하여 상수를 내놓거나 요약메모리에 접근한다.

if $(n \geq 0)$ ★ ⊤	if $(\text{lookup}_l \sqsubseteq \bullet)$ ★ ⊤	if $(n \geq 0)$ ★ lookup_l
(a)	(b)	(c)

그림 3.2: 검사하는 입력 예제

검사기는 다음과 같이 (a),(c)의 올바름과 (b)가 잘못 정의되었음을 확인해 준다. (a), (c)는 *Effect*가 ⊤이 아닌 반면 (b)는 ⊤이다.

$$\frac{\dfrac{\{\} \vdash n \Downarrow (\{\}, \bot) \quad \{\} \vdash 0 \Downarrow (\{\}, \bot)}{\{\} \vdash n \geq 0 \Downarrow (\{\}, \bot)} \qquad \{\} \vdash \bigstar \Downarrow (\{\}, \bot) \quad \{\} \vdash \top \Downarrow (\{\}, \bot)}{\{\} \vdash \text{if } (n \geq 0) \bigstar \top \Downarrow (\{\}, \bot)}$$

$$\frac{\dfrac{\{\} \vdash \text{lookup}_l \Downarrow (\{\}, \hat{\texttt{lkp}}) \quad \{\} \vdash \bullet \Downarrow (\{\}, \bot)}{\{\} \vdash \text{lookup}_l \sqsubseteq \bullet \Downarrow (\{\}, \top)} \qquad \{\} \vdash \bigstar \Downarrow (\{\}, \bot) \quad \{\} \vdash \top \Downarrow (\{\}, \bot)}{\{\} \vdash \text{if } (\text{lookup}_l \sqsubseteq \bullet) \bigstar \top \Downarrow (\{\}, \top)}$$

$$\frac{\dfrac{\{\} \vdash n \Downarrow (\{\}, \bot) \quad \{\} \vdash 0 \Downarrow (\{\}, \bot)}{\{\} \vdash n \geq 0 \Downarrow (\{\}, \bot)} \qquad \{\} \vdash \bigstar \Downarrow (\{\}, \bot) \quad \{\} \vdash \text{lookup}_l \Downarrow (\{\}, \hat{\texttt{lkp}})}{\{\} \vdash \text{if } (n \geq 0) \bigstar \text{lookup}_l \Downarrow (\{\}, \hat{\texttt{lkp}})}$$

제 4 장 차후 연구

사용자에게 분석 대상 쿼리 정보를 좀 더 구체적으로 받아서 적절한 쿼리 판별기 생성 시스템을 만들어 볼 수 있다. 이번 연구에서는 분석 대상 쿼리에 대한 쿼리 판별기의 핵심 부분은 사용자에게 입력으로 받았다. 사용자에게 분석 대상 쿼리를 받을 때 단순한 문법 표현식(expression)만 받는 대신 논리식으로 기술된 제약식(predicate)을 함께 받고 γ함수도 함께 받아서 쿼리 판별기를 생성해볼 수 있다. 제약식을 기술하는 언어와 판별하는 방법에 대한 고민과 연구가 필요하다.

이 연구 목표는 문맥 구분 분석(selective context-sensitive analysis)만으로 제한되어있는데 문맥 뿐 아니라 다른 분석 민감도로 확장해볼 수 있다. 선별적 문맥 구분 분석을 위한 예비분석(impact pre-analysis) 연구 아이디어는 문맥 구분 분석 뿐 아니라 관계분석(relational analysis)이나 흐름 구분 분석(flow sensitive analysis) 등에도 충분히 적용될 수 있는 기술이다. 이러한 다른 분석 민감도에 대해서도 예비분석을 수행하도록 생성기를 확장할 수 있다. 이러한 확장을 통하여 두 분석 민감도를 동시에 만족해야 검증되는 쿼리를 위한 예비분석을 쉽게 제작할 수 있게 된다.

제 5 장 결론

이 논문에서는 선별적 문맥 구분 분석(selective context-sensitive analysis)을 위한 예비분석(impact pre-analysis)을 생성하는 방법을 제시하였다. 예비분석을 매번 직접 구현하기에는 비용이 많이 든다. 본분석(main analysis)과 연관된 요약 실행 의미 일부 그리고 쿼리 판별기를 입력으로 하여 생성기를 만들 수 있음을 보였다. 또 그러한 입력으로부터 추출해야하는 정보를 정리하였다. 선별적 문맥 구분 분석을 위한 예비분석이 효율적으로 수행되기 위한 조건을 검사하는 방법을 제시했다. 제안한 방법을 통해 생성기를 제작하여 여러 다양한 분석에 대응해 선별적 문맥 구분 분석 기술을 빠르고 손쉽게 장착할 수 있다.

참고문헌

[1] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” in *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1977. pp. 238–252

[2] Patrick Cousot and Radhia Cousot. “Systematic design of program analysis frameworks.” in *Proceedings of ACM Symposium on Principles of Programming Languages*, 1979. pp. 269–282

[3] Patrick Cousot and Radhia Cousot. “Comparing the galois connection and widening/narrowing approaches to abstract interpretation.” in *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, Springer-Verlag, 1992. pp. 269–295

[4] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi, “Selective context-sensitivity guided by impact pre-analysis.” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014. pp. 49

[5] David A. Schimdt, “Trace-Based Abstract Interpretation of Operational Semantics” *Journal Lisp and Symbolic Computation archive* vol. 10, no.3, pp. 237 - 271, May 1998.

Abstract

This paper presents a way to generate impact pre-analysis for selective context-sensitive analysis based on abstract interpretation[1, 2, 3] and to check the correctness of the input for the generation. It costs a lot to design and implement impact pre-analysis for selective context-sensitive analysis because it consists of many algorithms. It is impossible to make general impact pre-analysis because the design of it depends on its target main analysis. The part depending on main analysis in impact pre-analysis is abstract semantic function. We can generate impact pre-analysis receiving a part of abstract semantic function as an input. We can make only impact pre-analysis' generated by correct inputs be performed by checking the correctness of the input's design. We can check the abstract semantic function given as an input using static analysis. This paper presents a core language describing an abstract semantic function given as an input and the analysis design for checking the correctness of the input.

Keywords: abstract interpretation, static analysis, program analysis, selective context-sensitive analysis, generation

Student Number: 2012-20724